07.2017

# Snapshot Ensembles (Extended Edition): Train 1, Now Get Even More

ALI JAD KHALIL

# Definition

### Project Overview

Parameters for complex machine learning models, like deep neural networks (DNN's), are generally optimized to minimize loss functions with a multitude of local minima. Though the model weights corresponding to these various local minima generally provide a comparable level of overall accuracy in classification problems, they each have their own biases in classify certain examples better than other examples. In other words, two sets of weights associated with two different, but equally accurate local minima will almost always disagree about the classification of at least some examples. Ensembles of models leverage this property to capture and smooth out differences in local minima of individual models. An ensemble will combine several, separately trained models with comparable test accuracies and average their softmax outputs. Normally, this technique results a considerable improvement in accuracy. However, this benefit of ensembles comes at the expensive of training time linearly increasing by a factor of the number of underlying models in the ensemble.

To address this problem of increased training time, Huang et al. suggest a new approach training an ensemble of models [1]. Essentially, their process divides the total training time for a single model into M sections. Each section is used to train one "Snapshot" model using a cosine annealing learning rate schedule with stochastic gradient descent (SGD). In this training technique, each subsequent Snapshot model settles at a local minimum with an accuracy closer (than the previous Snapshot model) to the final minimum accuracy of the model trained normally over the entire training period. By using this approach, training will conclude with M models each having their own unique local minima and an classification accuracy comparable to that of a normal training cycle. Collectively, these Snapshot models can be used to form an ensemble capable of significantly improving accuracy relative a single model without having to increase the allotted training time at all.

This project aims to extend and ideally improve the original Snapshot ensemble approach by examining its strengths and shortcomings. In particular, it will examine the characteristics of local minima of Snapshot models. Furthermore, once the key attributes contributing to performance have been established, the project will examine their potential to be exploited to find better local minima for combining in a Snapshot Ensemble. This should ideally bridge the gap in classification performance between a Snapshot ensemble and a regular or conventional ensemble of models. Through this process, we ultimately show that it is possible to improve on Snapshot Ensemble by tweaking them in subtle ways.

## Background Information

Whether in natural language processing (NLP) or image classification problems, accuracy is normally the bottom line metric in evaluating new models. As such, any improvements in accuracy given a model with a fixed number of parameters or training time is noteworthy. For that reason, the recent introduction of the "DenseNet" model has been particularly heralded [2]. It promotes feature reuse as well as effective skip connections to drastically shorten the path between any two layers in the network. As such, this architecture is currently the most efficient on a per-parameter basis for image classification.

Though an individual model like DenseNet is impressive on its own, empirically, ensembles of the same model architecture have long been able to boost classification accuracy. This has shown to be the case for multiple DenseNet models as well as practically every other convolution neural network (CNN) model. Accordingly, there has always been a special interest in underlying principles causing ensembles to be so effective. In particular, it has long remained somewhat of a mystery why a model trained two separate times often yielded two different local minima.

Im et al. aimed to explore and uncover some of the features of loss surfaces as well as their connections to various optimizers [3]. They showed that the selection of an optimizer alone results in sometimes radically different local minima. By even changing the optimizer at an arbitrary point late in training, it can produce a local minima with drastically different features to the one found without the change. In addition, different optimization loss functions inherently induce different local minima and in turn, different results. One such example is the triplet loss function. It has recently gained popularity in instances where connections between image classes may not immediately apparent in the picture; but rather, a class label could a function of something implicit in the image (like posture or positioning of several objects) [9].

Outside of changing a model's optimizer or loss function, another way to quickly find several of these differing local minima in a short period of time is by employing a cosine annealing learning rate schedule [4]. Essentially, it simulates a normal SGD schedule over a much shorter period of time. This expedites the convergence process considerably and is therefore well-suited for shortening the training time of a model. At the end of a cosine annealing schedule cycle, it can be repeated to dislodge training from the current local minimum and find a new one. Ultimately, this process is designed for the discovery of a number of new, useful, and accurate local minima in an abbreviated amount of time.

## Datasets

To simplify things and aid in isolating variables, the dataset in examination for the project is CIFAR100 [5]. It can be downloaded and almost immediately used with the Keras framework [6]. While not as famous as ImageNet, it is still a fairly hard dataset to achieve high classification performance and contains many classes. (The absolute best models with 40M+ parameters have a top-1 error rate of about 17 percent.) Traditionally, these multi-class dataset benefit most from the smoothing effect of ensembles. In fact, CIFAR100 is main dataset used in the original Snapshot ensemble experiment. It therefore provides context for any comparisons and extensions of the previous research.

**Problem Statement**

Empirically, the Snapshot ensemble methodology works quite well to improve accuracy with a limited time constraint. On the CIFAR100 dataset, irrespective of the model architecture of choice, the Snapshot ensemble improves the top-1 error rate by 15-20 percent (compared to a single model trained over the same period of time). However upon further inspection, the local minima discovered by Snapshot models with the cosine annealing learning rate cycles in a Snapshot ensemble are quite similar.

In the paper's initial experiment, they implicitly discovered this property of their local minima. First, they tried cosine annealing with the learning rate at 0.1. This resulted in local minima with a very low disagreement percentage and small distances between their final embedding activations. In fact, in their own graphs for Snapshot models with a learning rate of 0.1, it shows a marginal improvement in accuracy with the use of more than two underlying Snapshot models in a Snapshot ensemble.

After switching the starting learning rate to 0.2 for the cosine annealing process, the model weights were more severely displaced from their local minima at the beginning of each training cycle. This dislodgement in turn resulted in a greater disagreement percentage between the local minima. Therefore, the resulting local minimum of each model was more useful in the context of an ensemble. In fact, with an initial learning rate of 0.2, there was a meaningful dip in error rate for every additional Snapshot model added to the Snapshot ensemble (up to 4 underlying models).

As further underlying evidence of the issue, there was very little degradation of the model when two Snapshot models' weights were combined using interpolation. With the cosine annealing learning rate schedule, sequential models especially could fairly easily be mixed together without impacting the error rate. This was not the case with two models trained from scratch having random initialized weights. That distinction that true ensembles experience a much larger increase in accuracy with each additional underlying model in the ensemble.

By extension, it should therefore very possible to further improve the accuracy of a Snapshot ensemble by intentionally seeking out largely different local minima. This paper will explore ways of finding those contrasting local minima (while obviously maintaining the Snapshot ensemble's principle advantage of limiting the overall training budget to be no longer than that of a single model). In the end, it is the expectation that these diverse local minima will bridge that gap between the classification performance of a normal ensemble and a Snapshot ensemble.

**Expected Solution**

To achieve this goal of ameliorating the Snapshot ensemble, this project explored a plethora of methods to introduce variability into the local minima of the Snapshot models. One was tweaking hyper-parameters (like the optimizers and loss functions) during snapshot model training. As another more straightforward way of differentiating local minima of the Snapshot ensemble, it is possible to simply reset the initial weights more frequently. This approach must be balanced though with downside of reducing the individual Snapshot model accuracy too much by resetting the weights constantly.

In employing either of these changes to the original Snapshot ensemble, it is not absolutely necessary to observe an increase in the accuracy of each underlying Snapshot model. (In fact, it is likely that the accuracy of each individual Snapshot model may decrease slightly.) But rather, based on past research suggesting it is possible to induce drastic changes to loss surface convergence using just subtle changes in hyper-parameters, the goal is to simply elicit a higher of the level disagreement between the predictions of Snapshot models. As long as the accuracy remains comparable to previous Snapshot models, then the larger level of disagreement between models should result in a more accurate final Snapshot ensemble.

**Metrics**

At a high level, the first category of metrics focuses on numbers assessing the model’s ability to classify images correctly. Accuracy (as percentage of correctly classified images) is typically used a main barometer of the success of a CNN. More specifically, in this project, accuracy percentages express a model's ability to either classify an image with the correct label as its top prediction or alternatively, as a top-5 prediction (out of however many possible classes of images exist). During training, accuracy is represented more granularly for optimization purposes as a cross-entropy loss (e.g. essentially the negative log probability predicted by a model for the correct class). As an alternative loss function, a triplet loss value is optimized so that the final layer's embeddings in the CNN are close (in terms of L2 distance) for similar classes. These two loss functions can be combined in various ratios to explicitly alter the bias of a model’s predictions.

As the second general category of metrics, this project leverages a set of values to gauge differences in the local minima found during training. To estimate the similarity of two models (with the same architecture), you can linearly combine their weights using a technique called "interpolation".

Essentially, it is a weighted average between the two models' weights at each layer to form a new, single model. For models having converged to radically different local minima, this technique will produce a resulting model incapable of classifying images well. Conversely, models with similar local minima can be merged to produce a resulting model sometimes capable of classifying images equally as well or even better than each model individually. As for another less explicit technique of comparing local minima, it possible to take the average L2 distance between two models' final layer embeddings for all the test images. Local minima producing similar results will often have a shorter L2 distance between their final layers' embeddings. Finally, the easiest method for conceptualization is calculating the disagreement percentages between predictions of models with similar accuracies. Ideally in a Snapshot ensemble, it is advantageous to have models with a high level of disagreement. That way, when a model is incorrect, another model is more likely to not have repeated the same mistake.

# Analysis

### Data Exploration and Visualization

In this particular project, there was no need for extensive data exploration or analysis. The seminal paper for Snapshot ensembles leveraged the CIFAR10 and CIFAR100 datasets, so it was a matter of picking one of the two options. And since CIFAR100 was more difficult and therefore suited to the use of an ensemble, it was the choice.

CIFAR100 is an established dataset of 60,000 32x32x3 RGB images labelled into 100 different classes. They are pre-split into 10,000 test images and 50,000 training images. (In the training set, each class contains exactly 500 images.) Examples of images include airplanes, mushrooms, dogs, and castles and can be seen in Figure 1 as well as on the CIFAR100 website [5].

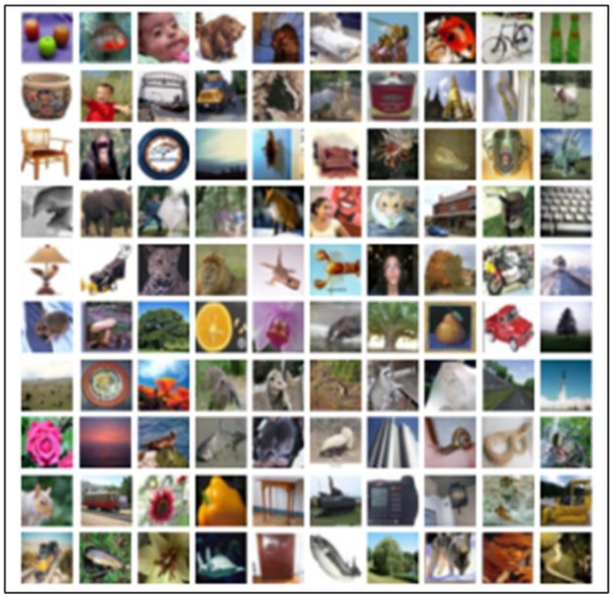
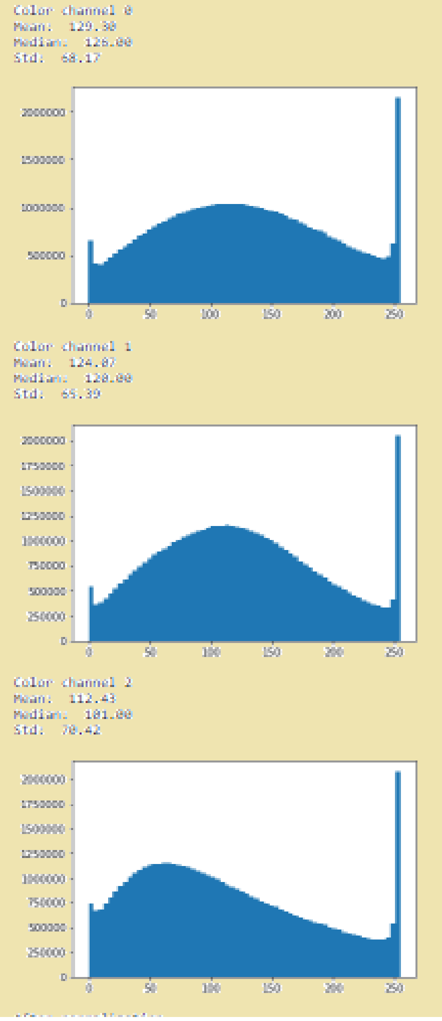


*Figure 1. Example images from the CIFAR100 dataset.*

Though the dataset does not have as many examples as ImageNet for instance, it has proven to be large enough to get a top-1 accuracy in the mid to high-80 percent range. This makes sense given that there are only 100 classes (instead of 1000 class with ImageNet). This smaller dataset is also useful because it allows iteration through training models at much faster pace. Due to the dataset's ubiquitous nature, it has been used for research quite frequently and often comes available with prepackaged code in many deep learning frameworks like Keras.

In terms of abnormalities, this dataset is fairly normal.  When breaking it down in terms of its RGB pixel values on a per-channel basis, each color channel is slightly skewed right with a disproportionate number of pixels having the value 255.  (The individual pixel values range in value between 0 and 255, inclusive).  With that said, the rest of the distribution for each channel is approximately normally distributed with median values between 100 and 120, respectively.  The distribution and basic statistics (i.e. mean, median, standard deviation) associated with each color channel can be found in Figure 2.



```
Color channel 0
Mean:  129.30
Median:  126.00
Std:  68.17
```

```
Color channel 1
Mean:  124.87
Median:  128.00
Std:  65.39
```

```
Color channel 2
Mean:  112.43
Median:  101.00
Std:  70.42
```

*Figure 2. Training set channel-wise data distribution (before any preprocessing).*

## Algorithms

### Convolutional weights

- Convolutional weights are the default weight structure for DNN’s with images and videos as inputs.  In fact, they are the defining characteristic of CNN’s.  Essentially, these neural network weights are applied to patches of images to extract features from successive pixels or groups of pixels in images.  This particular type of weights has proven to profoundly successful in variety of image-related tasks such as classification, segmentation, and detection.

### Batch Normalization

- Batch normalization (BN) is a technique employed to speed up training and regularize the network’s activations slightly (by subtracting a layer’s activations by the batch's mean activation and then dividing them by the batch's standard deviation) [7].  At interference time, instead of a particular batch’s average, BN leverages the running activation mean and standard deviation of a particular layer.  This allows the network to make each layer’s activations about 0 centered and standardized.  However, if those properties are not desirable, the network has several parameters to reshape the data back to its old form.  These parameters are called gamma and beta and are initialized to 1 and 0, respectively, by default.

Max pooling

- Because a typical CNN will get more features as the number of layers increases, max pooling is periodically used to reduce the spatial size of a layer's activations. To do that, the technique involves picking small, congruent spatial patches of each feature in a layer in the network and eliminating all activations except the maximum one. Normally, this will have the effect of reducing the number of activations by a factor 2 or 4 and ensures that the number of activations (e.g. the size of the model) in future layers is manageable. At a high level, this method is designed to reduce the spatial resolution of the model to compensate for the growing number of features.

Rectified Linear Unit (ReLU)

- To introduce non-linearity to a DNN so it can be truly expressive, rectified linear units are added as an activation after each convolutional layer [8]. They drop any activations with values less than zero and consequently, eliminate any gradient flow to those activations. Without them, the entire model will be linear, and as such, every activation in the model would be capable of being expressed with a single linear function.

Skip Connections

- Skip connections have been leveraged in recent model architectures to provide a direct path from one layer to another layer further down the network, usually via a summation of the two layers. This is useful in eliminating the vanishing gradient problem associated with a purely multiplicative path between two distant network layers. Therefore, these skip connections improve the models ability to train efficiently. The DenseNet model leverages a number of skip connections in each "Dense" block to improve gradient flow and allow the model to reach higher classification accuracies.

1 x 1 Convolutions

- At the end of each Dense block in the DenseNet model, there is a 1x1 convolutional layer. These layers perform a transformation on the existing to features to make them richer and more expressive; but also, the 1x1 convolutional layers function to reduce or compress the existing features into a smaller set of features. In turn, this reduction of features can significantly downsize the number of parameters in the model and make the model much faster to train without materially compromising accuracy. In other words, the 1x1 convolutions make the model more efficient on a per-parameter basis.

Cross-Entropy Loss

- A mathematical function used in the model's optimization process during training. It quantifies the accuracy of the model in its predictions of the correct class of an image. With one-hot vectors as the class labels, the cross-entropy value is just the negative log of the model’s predicted probability of the correct class. So if it is predicts a very low probability for the actual image class for example, it will produce a high cross-entropy loss. This particular loss function is extremely common, especially for models trained to perform image classification.

### Triplet Loss

- This loss function is often used to project faces into an embedding space so that the embeddings of faces of the same identity are close by each other. It is defined as the maximum between 0 and the L2 distance between an anchor image's embeddings and the embeddings of an image from a different class minus that same anchor image's embeddings with the embeddings of an image from the same class. Normally, the goal is keep a "margin" distance between the embeddings of images from different classes. It works very well in cases with thousands of classes to avoid using a softmax cross-entropy loss with too many possible options. Also, it is useful in instances where members of the same class aren't necessarily visually similar. But rather, they share some implicit connection. Some examples would be scoring a goal in soccer versus scoring a goal in lacrosse or a robot engaging in the action of picking up a ball and a human doing the same thing. Though not well suited for this specific image classification problem, the use of a triplet loss function is still practical (when mixed with a cross entropy loss) in causing the model to capture unique and hidden features.

### Data Augmentation

- To improve the generalization ability of a model, images are altered before training so that they are randomly flipped, rotated, zoomed, and cropped. Especially with smaller datasets, this technique, called data augmentation, prevents overfitting and in essence generates new images (based off the original ones). When used during training, the method improves test accuracy so much that it is used as training technique in practically every image classification model today.

### Cosine Annealing Learning Rate Schedule

- To speed up training time, the use of a cosine annealing learning rate schedule (along with SGD) has proven to be an effective [4]. In a matter of 30-50 epochs, it is possible to approach accuracies that would otherwise take 200-250 epochs during training with a more conventional learning rate schedule. If cosine annealing is done in cycles successively (so that the learning rate is restart to the initial learning rate after a set number of epochs), the local minima found will be different and therefore contain some disagreement in their predictions. For those reasons, this training technique is essential in constructing a Snapshot ensemble with varying local minima in each underlying Snapshot model.

Late Switching of Optimizers

- As examined in several studies, local minima change greatly based on the optimizer used during training. In fact, it is possible to severely alter the final local minima found at convergence by simply changing the optimizer at a late stage of training [3]. Using that late switching methodology, it should be possible to inject greater disagreement into the underlying models in a Snapshot ensemble. And by virtue of that, it will increase the overall test accuracy of the ensemble.

### Benchmarks

The benchmark for this particular experiment is the original Snapshot ensemble for CIFAR100. According to the experiment, it featured 5 Snapshot models with each trained using the cosine annealing learning rate schedule for SGD. Instead of using the values from the paper, the experiment was conducted using my particular 2 million parameter DenseNet implementation. This choice ensured consistency in terms of the number of parameters and architecture of the network in each underlying Snapshot model across all variants of Snapshot ensembles. That way, each Snapshot ensemble could be trained for exactly the same number of total epochs, and the resulting final accuracies would be comparable. With that said, in this paper's recreation of the original Snapshot ensemble with 5 Snapshot models, it achieved a top-1 percent accuracy of 79.75 and a top-5 percent accuracy of 95.38 on CIFAR100.

## Methodology

### Data Preprocessing

Given the widespread use of the CIFAR100 dataset, Keras offers a function to download the data (if not already cached on the system) and return it as a tuple of training images and labels and validation images and labels. As previously mentioned though, DNN's tend to better handle normalized and standardized data. So each color channel in both the training and test dataset was subtracted by the training set’s channel mean and divided by 128. That way, each channel’s pixel values would be roughly centered at 0 and range from -1 to 1. Although this doesn't exactly adhere to the traditional steps for normalizing or standardizing input data, it is mixture of the two techniques and in practice worked quite well.

Additionally, the dataset only contained 50000 training images. That dataset size is relatively small given the inordinately large number of parameters in today’s DNN. Therefore, to avoid overfitting and improve generalization, the images were also pre-processed with common image augmentation techniques.  More specifically, input images were randomly rotated, cropped, flipped, and enlarged using a Keras class called "ImageDataGenerator".  With these alterations to the input data, the dataset is effectively made larger by introducing new off-shoots of the original images.  In almost all cases, these image augmentation techniques have boosted validation accuracy.

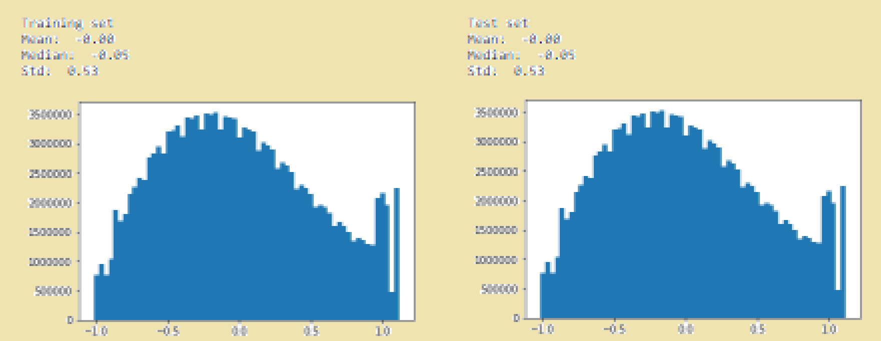


*Figure 3.  Distribution of training and test set data after standardization and normalization.*

Finally, while the input data was skewed to the right in some color channels because of the large number of pixels with the value 255, that property is not really correctable using data transformations.  When data is gradually skewed right or left, it is possible to take the log of the data and make it to look slightly more Gaussian in nature.  However, in this case, that technique is not possible, so the disproportionate number of pixels with a value of 255 is an accepted property of input data in most image processing networks and CNN's.  Aside from that, there were no real abnormalities in the data, so there was no need to take any other measures during preprocessing to address for inconsistencies in the data.

### Implementation Details

The code for the project was split into 3 separate files - "snapshot_train.py", “trips_util.py”, and “snapshot_eval_utils.py”.  Each file was thoroughly documented and featured a unique subset of functionality.  In summary, the “snapshot_train.py” file contained the main routine and core high-level functionality for the project.  To avoid making the main file too long and for the sake of instituting modularity in the code, the “trips_util.py” and “snapshot_eval_utils.py” files supported the main routine by having the low-level logic for training DNN’s on a triplet loss function and also evaluating the Snapshot ensemble after training was complete.

As previously mentioned, the “snapshot_train.py” file contains the high-level functions for training a simple Snapshot model as well as a Snapshot model with a triplet loss component.  As a necessary part of that type of training, the file featured the cosine annealing learning rate scheduler function to expedited normal SGD.  The main routine of this file first creates lists of sequences of hyper-parameter changes in each test Snapshot ensemble.  It then iterated through each sequence, trained the underlying Snapshot models, and saved their weights and histories.  At the end of this process, every unique Snapshot ensemble (composed of a sequence of hyper-parameters) could be recreated and evaluated using their respective weight and history files.  In addition, this Python script contained code to provide the option of training a single normal model.  This model serves as a reference point for a normal training routine with a fixed 155 epoch training period on my DenseNet implementation.

The project also required a whole suite of functions and classes for training models with a triplet loss function.  During triplet loss training, input images are presented in sets of three images at a time.  More specifically, there is an anchor image, a "positive" image (from the same class), and a "negative" image (from another image class).  For this type of training, it is imperative to select examples where the anchor image's final layer embeddings are closer to the negative example's embeddings than the positive example's embeddings.  (That way, the model can observe example images with undesirable embeddings and learn from them.)  To do that, the project features a generator capable of producing randomly selected easy, semi-hard, and hard batches of triplets.  Normally, triplet loss training would begin with easy examples (where the anchor and positive examples have somewhat close final embeddings).  Then as the model becomes more fine-tuned to the loss function's objective, the generator can be passed an argument to select progressively harder triplets.  Obviously to determine the difficult of triplets, there needed to be a function to get the model's current final layer embeddings for all the training images.  These normalized layer embeddings would be recalculated every couple epochs to ensure that the triplet selections were still accurate in terms of their difficulty.

In addition to the generator, triplet training in Keras requires the implementation of custom loss functions.  As the first one, the "margin" loss function tried to maintain a maximum margin distance between the positive and negative example images’ embedding layers.  In this project, for L1 distance margin function, the project's distance value was 0.6; and for the L2 distance margin, the distance was set to 0.3.  Additionally, for the categorical cross-entropy portion of the loss of function, triplet training used a customized entropy function factoring in only the anchor's cross-entropy loss value.  This decision reduced redundancy in the images factoring into the loss, because it was trivial to limit the number of triplets selected for each random anchor.  However, it was far more complicated to control how often a distant positive example for instance would be reappear with a particular anchor and different negative example.  In cases where the generator was attempting to produce semi-hard or hard examples, a positive image with a large embedding distance from its anchor would be selected very frequently and therefore skew the gradient updates to over fit to that specific example.

As the final component of the source code, there is an entire section of functions designed to aid in the evaluation of Snapshot models and ensembles.  As one example, there is a function designed to save basic evaluation metrics (e.g. the accuracy and loss values) for each Snapshot ensemble.  This function enables a user to discern the best Snapshot ensembles.  Using those selected ensembles, there is another function for saving advanced metrics for a particular ensemble.  More specifically, this function shows and saves the L2 distances between the underlying models’ final embedding layers, basic metrics of an interpolation of the weights of the underlying models, and lastly, the disagreement percentages between predictions of the underlying models.  By saving these statistics, it helps to avoid recalculating them later, understand the ensembles’ local minima, and generate graphs at a later stage.

This evaluation code also includes printing functions to display relevant metrics and values. For instance, there is functionality to print the histories of each underlying Snapshot model in an ensemble. In addition, there is a function to print exclusively basic evaluation metrics (such as loss, top-1 and top-5 percent accuracy) when the Snapshot model is created using an arbitrary set of underlying models. And finally, there is code providing the ability to display the L2 distance embedding distance between each Snapshot model, interpolation results, and disagreement percentages (assuming that these value have previously been calculated and saved). Without these functions, it'd be virtually impossible to make the assessments needed to ultimately improve various Snapshot ensembles.

**Coding Complications**

As one of the first complications encountered in the coding process, there was a problem in generating triplets for the triplet loss training. While it would have been ideal to use the model's current weights to produce the final layer embeddings needed for selecting image triplets, it was not possible. Since the models weights are constantly being updated during training, they are not usable for inference without causing an unrecoverable threading error. Theoretically, the model weights could have been copied to another template model at the end of each batch, and the new embedding calculations could be done using those weights. However, even that idea did not work with the TensorFlow backend. For a solution, the embeddings for all available input images were calculated before instantiating the generator and then were simply passed to the generator. This worked well in practice (due to the small size of the training set) and would take place once every 2 or 3 epochs.

As another issue related to the triplet batch generator, it was quite slow in the initial implementation. However, by making it multi-threaded, the speed improved significantly. Later, it was discovered that Python's multiprocessing was significantly more efficient than the multi-threading one for whatever reason. So, I then decided to use that module instead and saw a 5x speed up for of triplet selection relative to the initial single-threaded code.

As the next issue, the "Adam" and "Nadam" optimizers in Keras were surprisingly sensitive to their initial learning rate. If the learning rate was set too high, the learning would plateau very early on in training. If it was too low, training would not progress fast enough. This also proved to be true for the "Adadelta" optimizer as this same problem was observed when training a normal model using Adadelta. Ordinarily, this level of learning rate sensitivity should not apply for adaptive optimizers like Nadam, Adam, and Adadelta.

Finally, though not a complication related to coding specifically, it was very noticeable from the start that training would take a long time on my two GPU system. Therefore, it seemed prudent to have half of the Snapshot ensembles training on one GPU and the remaining half on the other GPU, concurrently. To do that, the main Python routine would be run with "CUDA_VISIBLE_DEVICES=<GPU_number>" as an environment variable for each half of the set of ensembles. That way, the CUDA C++ library (controlling the GPU functionality in Keras/Tensorflow) would only use a specific GPU during training.

**Refinement**

Initially, it was suspected that alternative optimizers like Adam and Adadelta could replace SGD with a cosine annealing learning rate schedule during the training of some Snapshot models in an ensemble to produce more variety in the local minima. During this project though, it was quickly discovered that was not possible. For one, SGD with cosine annealing trains and reaches a competitive local minima much faster than other optimizers. Secondly, its design allows it to dislodge a previous Snapshot model's local minimum and still quickly find another accurate local minimum. This ability to both displace the current training and then quickly settle at a new quality local minima does not seem to apply to other optimizers.

It was then determined that it would be more effective instead to augment the SGD with a cosine annealing learning rate schedule in a couple of ways. By injecting a new term into the traditional cross entropy loss function and still using cosine annealing SGD, it yields Snapshot models with vastly different local minima. Additionally, though not as successful because it turned out that switching optimizers required at least 50-100 epochs with each optimizer to observe the impact on the local minimum, it was also possible to switch from SGD to another optimizer, like Adam, with an extremely low learning rate for the last 30 percent of a Snapshot model's training. These alterations, to varying degrees, were found to be competitive with and sometimes improve the performance of the original Snapshots ensembles (trained exclusively with SGD cosine annealing and a vanilla softmax cross-entropy loss function).

While the triplet loss-based Snapshot models did provide diversity to the ensemble's local minima, the pure triplet loss function did not immediately work particularly well in this context of this project. It required placing a heavy weight on cross entropy portion of the loss for it to be competitive with a normal cross-entropy loss. From a theoretical standpoint, it was obvious that the triplet loss was not well suited for pure image classification. Also, because of the poor fit of the triplet loss function, it was important to select triplets from the pool of every possible training image so that the model still got exposure to the entire dataset. During the selection of hardest possible triplets, it is common that the triplet generator cycles through very small subset of the training examples. To compensate for that, it was necessary to configure the generator to periodically select a random triplet pair. Also, part of the time, instead of selecting the hardest example, the generator would relax that requirement slightly. Together, these policies of triplet selection helped make it more compelling in juxtaposition to a Snapshot ensemble with only one loss function.

However, because the difference in performance was still not huge, further examination and experimentation was necessary. It revealed that there were diminishing returns for Snapshot ensembles containing more than 3 or 4 models. In theory, this is due to the similarity between the local minima of the Snapshot models trained in succession. Therefore, by restarting the initial model weights once for every 2 or 3 Snapshots models, minimum, it was possible to improve the overall accuracy of the ensemble. (The weight reset introduces a new set of radically different local minima in the next group of Snapshot models.) For longer training periods, this classification improvement was especially noticeable. In particular, a Snapshot ensemble enough training time for 6 or 7 Snapshot models benefits significantly from at least one or two weight resets.

Therefore, using the knowledge acquired through trial and error during the project, the project settled on several final solutions capable of clearly improving on the original Snapshot model. As one example, the total training time for the benchmark Snapshot ensemble with 5 Snapshot models was split into 3 parts. Because more than 3 or 4 models in an ensemble empirically did not seem to improve accuracy too much, it made more sense to limit the ensemble to only 3 models and use that increased training time per model to simply reset the weights after each cosine annealing cycle. Other ensembles followed this same logic except one or more of their Snapshot models used a mix of loss functions. Also, since the original Snapshot paper had success in dislodging training from its current local minimum by going from an initial learning rate with cosine annealing of 0.1 to 0.2, the project took things one step further. By starting the first cycle of the Snapshot ensemble at 0.2 and gradually increasing each cycle's initial learning rate by 0.025 for each of the 5 Snapshot models, the last Snapshot model would use an initial learning rate of 0.3 and in theory, be further dislodged from the preceding local minimum.

# Results

### Model Evaluation and Validation

The original Snapshot ensemble took a normal model's training period and rededicated it to training five Snapshot models to ultimately form an ensemble. This methodology was largely successful because the combination of distinct local minima in the Snapshot models smoothed out the biases of each individual Snapshot model and boosted test accuracy considerably. However, the extent of the smoothing effect is directly correlated to the variation in the local minima of the underlying models; and in the original Snapshot ensemble, its models' local minimum are unfortunately quite similar. It is therefore obvious that any final solutions to the problem of maximizing model accuracy with a limited training budget will have to improve local minima variability without sacrificing underlying model accuracy too much.

More specifically, this project found four final solutions meeting these expectations by capably adding entropy to an ensembles' collection of local minima without torpedoing its underlying model accuracy to ultimately improve its overall classification abilities (compared to the benchmark ensemble). As the first solution, the modified Snapshot ensemble required simply lengthening the training time of each underlying Snapshot model so that instead of five Snapshot models, there were only three. This had the effect of enhancing the average accuracy of the three underlying models by 0.75 of percentage point and in turn, boosting the overall accuracy of the ensemble. Next, this idea was extended by leveraging the increased training time for each of the three models time to add diversity to the local minima. More specifically, after each of the three Snapshot models, the model weights would be reset. This was the best performing Snapshot ensemble with a top-1 accuracy of 80.3 percent – beating the original one by over 0.5 of a percentage point. As another Snapshot ensemble following this same high level reasoning of adding entropy to the local minima, the third proposed ensemble used all five of the training periods, but rather than restarting the learning rate to 0.2 at the beginning of each cosine annealing schedule, it would gradually increase the learning rate's restart value and at same time, reduce the length of each training cycle. This also worked to boost the overall top-1 accuracy of the Snapshot ensemble despite featuring a lower average accuracy per underlying model. And finally, the last solution involved an ensemble with one of its models using the triplet loss function. While this ensemble idea was not superior in terms of accuracy to the original Snapshot ensemble, it was very close and functions mostly as a placeholder for the use of any alternative loss functions, or even different types of input data like optical flow (rather than images) for action recognition. It is believed that when deployed appropriately, the use of an alternative loss function could theoretically provide the biggest improvement to accuracy.

Although the exact configuration of each solution ensemble may not be directly usable to every machine learning problem, the high level ideas are robust and with slight adaptations, will almost always result in a performance boost over the original Snapshot ensemble in any deep learning problem. In particular, the weight reset recommendation can be extrapolated to any problem. Looking at Figure A1 as a general guideline to dividing up a fixed training budget, it is especially useful to reset weights more frequently in experiments with longer training budgets. As evidence, fully trained normal ensembles of models are significantly more accurate than Snapshot ensembles (due to the random starting weights for each model leading to more diverse local minima). This weight reset technique when carefully applied can even be useful for shorter training times. In this project with a very short training budget, each of the four final Snapshot ensemble alternatives have larger average distances between their final layers' embeddings than the original Snapshot model and still maintain a competitive average underlying model accuracy. Whether via a weight reset, careful selection of the number of underlying models, or an increase in the starting learning rate of each cycle, it is almost always beneficial to the ensemble to get higher levels of disagreement and distance between local minima even it comes slightly at the expense of accuracy for each of the underlying models.

Lastly, although the triplet loss model was particularly sensitive to this problem (as well as any other generic image classification problem), the strategy of using alternative loss functions in ensembles is worth examining more abstractly for machine learning tasks such as facial verification or action recognition. While for this project, the triplet loss model showed its merits and would likely have been more useful to its ensemble with a larger training budget, in a different context more suited to two different loss functions, it would excel even more. It is possible to replace any individual or string of underlying Snapshot models from the table in Figure A1 with models trained using SGD with a cosine annealing learning rate schedule and a new type of input or loss function. This replacement technique in good use cases will almost certainly improve overall accuracy over an ensemble with each model having been trained using the same loss function. As proof, it is only necessary to look as far as Temporal Segment Networks [10] and the Two-Stream RNN/CNN [11] as examples of ensembles benefitting from its underlying models being trained with distinct input and objective functions. It's why this suggestion of using alternative inputs and loss functions should not only be trusted, but researchers should actively be looking for ways to enhance their ensembles of models with this approach.

**Justification**

Despite it being a consequence of introducing local minima diversity to an ensemble that the new solutions' underlying Snapshot model accuracies were often worse than the original Snapshot ensemble's underlying model accuracies, it was necessary to keep them competitive by using the cosine annealing learning rate schedule. This learning rate schedule drastically sped up convergence time of a single Snapshot model so that there would be enough time to train several Snapshot models within the training budget. Without it, there would not have been a sufficient number of quality underlying models to form an ensemble. And though using other optimizers may add more variability to the local minima in an ensemble, it did not justify the enormous loss in underlying model accuracy due to insufficient training time for the alternative optimizer. For that reason, every single solution was based around the SGD optimizer with the cosine annealing learning rate.

But to push the success of the project solutions further, the key was restarting the weights in a logical fashion. It adds entropy to local minima in a similar fashion to the approach for standard ensembles. However, even though variability in the local minima of the underlying models is the best source for extreme boosts in ensemble accuracy, it is not efficient to simply reset the weights at the end of each training cycle. Therefore, resets must be mixed in appropriately so the underlying models in the Snapshot ensemble can maintain similar accuracies to the original Snapshot ensemble's underlying model accuracies. This means sometimes conducting multiple cycles of the SGD cosine annealing without a weight reset. As training time increases though, it is possible to mix in more and more weight resets to add diversity to the ensemble's local minima.

As further justification of these decisions, the statistics and graphs clearly indicate the rationale behind the construction of these final solutions is sound. Starting with the Snapshot ensemble simply lengthening the training time of each model and reducing the number of models, its average accuracy for each underlying model (Figure A6) is significantly higher than the original Snapshot ensemble's average model accuracy, and it only suffers very slightly in its average prediction disagreement (Figure A5) and distance between final layer embeddings (Figure A3). As a result, it makes sense that it would be an improvement especially given that the original paper showed marginal improvement for ensembles with more than three models. Approaching the problem in a different way, the ensemble with weight resets at the start of each of the three training cycles actually had the lowest average underlying model accuracy (Figure A6). However, looking at interpolation error rate, average prediction disagreement, and embedding distances, it had easily the highest average for each of these metrics. That extreme local minima diversity ultimately won out as this ensemble had the highest ensemble top-1 accuracy percentage. With the ensemble using the increasing learning rate restart values and progressively shorter training cycles, the interpolation error rate (Figure A4) and average prediction disagreement (Figure A5) are much higher than the original Snapshot ensemble. Like the original Snapshot ensemble, the weights are left untouched between training cycles; but the increasingly higher learning rates in conjunction with the shorter training periods seem to cause a larger level of displacement between the current model's local minimum and the previous model's local minimum. Lastly, the Snapshot ensemble with the triplet loss Snapshot model has a large embedding distance between the triplet model and its predecessor despite continuing to use the previous model's weights. This L2 distance is actually 0.1 units greater than the distance between the ensemble's first cross-entropy loss Snapshot model and the last underlying cross-entropy model with weights reset in between them (Figure A3). This indicates that by using a different loss function, the triplet loss model may be finding different types of solutions from the models with a standard cross-entropy loss.

Having said that, it is clear that the final solutions and the accompanying implicit guidelines for generalization (in Figure A1) did adequately solve the problem by improving on the benchmark for Snapshot ensembles in a variety of ways and demonstrating the potential for even further improvement with more training time. Moreover, in a different context, the integration of new input data and loss functions into an ensemble could boost performance even more than an increased training budget. So despite not showing an immediate drastic improvement within the framework of this particular experiment, the design of this project was always skewed towards the incumbent Snapshot ensemble. There was never expected to be any kind of monumental leap in accuracy. However, given that there was even a jump in ensemble accuracy at all, the high level implications of these findings are undoubtedly profound and clearly show the path to bridging the gap between a normal ensemble and Snapshot ensemble.

# Conclusion

### Free-form Visualizations

Please refer to the "Appendix" section for the free-form figures and graphs. These items are explained in further detail both in that section and also in the "Justification" section above.

As a general note though, it is important to explain the formatting of the labels (describing the sequences of hyper-parameter choices) for each of the Snapshot ensemble solutions in the graphs. For example, the Snapshot ensemble using a triplet loss function is expressed as “[r52sgd200, c52triptwo200, r52sgd200]”. Each token in the sequence represents a single underlying Snapshot model. More specifically, the first token – r52sgd200 – refers to the first Snapshot model in the ensemble. This particular model was trained for 52 epochs using the SGD cosine annealing schedule with a starting learning rate of 0.2 (e.g. the last number divided by 1000), a categorical cross-entropy as the loss function, and a random initialization of the weights (e.g. indicated by the first “r” character). Likewise, the second token -- c52triptwo200 -- refers to the next Snapshot model in the ensemble. It was trained for 52 epochs using the SGD cosine annealing schedule with a starting learning rate of 0.2, a loss function with an L2 triplet loss component, and the initialization of the weights being a continuation from the previous model (e.g. indicated by the first “c” character). In total, this particular Snapshot ensemble is comprised of three underlying models – one for each token.

## Reflection

In the end, the project’s optimal solution involved lengthening the training time of each Snapshot model at the expense of the number of models in the Snapshot ensemble. The added training time per model with this approach allowed for a weight reset after each cosine annealing cycle and therefore increased local minima diversity. The ensemble reached a top-1 percent accuracy of 80.32 percent and a top-5 percent accuracy of 95.4 percent. So while this particular solution may not have been markedly better than the benchmark Snapshot ensemble, it is still notable because when training for only 155 epochs, it is supposed to heavily favor the original Snapshot ensemble. With a small amount of training time, the number of unique and quality local minima found in the original Snapshot ensemble normally outweighs the diversity of local minima in an ensemble fewer underlying models.

Examining that point a bit further, the choice of total training time for the ensemble was critical. Because the original Snapshot ensemble is best suited in instances with limited training time, it was always the goal to devise mechanisms to improve on the accuracy of an original Snapshot ensemble in that short training period. With more training time, the techniques uncovered in this project should only widen the gap in performance from the original Snapshot ensemble. Furthermore, by selecting a shorter training duration, it allowed for the iteration through many alternatives and ideas during the project. Though theoretically, some ideas, like the switching of optimizers, should have worked better to improve local minima diversity, it was only through testing that there could be any kind of concrete or definitive validation about the actual usability of these new ideas.

At a high level, this project seems to really highlight the dichotomy between training speed and final model performance. On the one hand, a simple cosine annealing SGD learning rate schedule can reach a decent approximation of the final accuracy of a fully trained model within just 30 to 50 epochs. On the other hand, by separately training several models with unique weight initializations, it is possible to significantly boost the final performance of a single model. In between though, there are varying levels of accuracy given a trade off with training time. With a very limited amount of training time, a Snapshot model with 3 cycles and no weights resets may be best. With slightly more training time, it is better to reset weights occasionally after several cosine annealing cycles to introduce variation in the local minima (and also maybe throw the first Snapshot model after a reset).

As maybe the main contribution of the paper though and an outlier to the analysis concerning the tradeoff between time and accuracy, the infusion of new loss functions or inputs seems motivate the model to learn different information and features about a problem. In this project, the set-up and classification objective were not tailored to fully exploiting that benefit of an alternative loss function. However, the project managed to make some grounds with a mixed loss function and at least shed light on the potential of infusing multiple loss functions and training methodologies into a Snapshot ensemble for improving the final accuracy.

### Potential Improvements

In term of implementation improvements, it would be ideal to have a function for the triplet selection generator to calculate the model's final layer embeddings in real time with updated weights. This could be done by loading the weights after each batch to a new model localized to a specific, unused processing unit. That way, there would be no concurrency issue on the GPU attempting to get the embeddings. Alternatively, by looking at the backend Keras or Tensorflow code, it would reveal why exactly this particular functionality does not seem to work. Using that information, it would be possible to update my code or the back-end code with a hack to circumvent the problem.

This improvement in the generation of embeddings would make triplet training much more efficient in terms of its speed and ability to optimize parameters for the triplet loss function. Now, the triplet generator never has access to up-to-date embeddings from the model so its selections of triplets may get progressively easier until it gets an update to the embeddings. At which point, the triplet loss value will shoot up because the model will then see the actual hardest embeddings from the image dataset.

As a second potential improvement, instead of focusing mostly on an immediate extension to the Snapshot ensemble, it may be better to try shortening the training time of a single normally trained model by 30 percent. With that additional 30 percent of saved time, it would then be possible retrain the last the 25 percent of the model (trained for a shortened period of time) with three different optimizers to find new local minima. Overall, this would only increase the total training time by 25 percent and maybe result in a significant improvement to the Snapshot ensemble method.

Though there is no guarantee that this idea would materialize into an improvement over the Snapshot ensemble, it would still be interesting to actually apply the discoveries of Im et al. regarding the impact of changing the optimizer on the final local minima. This process would require training a normal model (with SDG as the optimizer) to completion and then taking the final 50 epochs or so and retraining them with a new optimizer to get a new and radically different local minima. Based on the papers, SGD-RMSprop, SGD-SGD Momentum, and SGD-Adam (to a lesser degree) seem to be most compelling choices for optimizer transitions.

Finally, as a last potential improvement, it may make sense to remove the batch normalization layers from the DenseNet model architecture used for training of the Snapshot models. There is countless research showing that BN is highly robust to initial weights in finding a local minima and therefore may not promote as much diversity in the predictions of successive models. So by removing the parameters associated with BN, it may be prudent to use that saved training time to add more convolutional parameters to each Snapshot model or instead simply allocate more training time per Snapshot model to achieve a better accuracy per model.

Though BN has been found to improve training speeds in terms of time to convergence, it may have been detrimental in the assessments of the effect of changing hyper-parameters between Snapshot models. Without batch normalization, "loss surfaces between different local minima are more complex in appearance" and may therefore have been more conducive for use in ensembles [3]. In this case, because of the speed and robustness of the cosine annealing learning rate schedule, batch normalization's benefits are marginalized and even become weaknesses due to the overarching desire for differences in the final set of local minima in the Snapshot ensemble.

## Sources

1. https://arxiv.org/pdf/1704.00109.pdf

2. https://arxiv.org/pdf/1608.06993.pdf

3. https://arxiv.org/pdf/1612.04010.pdf

4. https://arxiv.org/pdf/1608.03983.pdf

5. https://www.cs.toronto.edu/~kriz/cifar.html

6. https://keras.io/

7. https://arxiv.org/pdf/1502.03167.pdf

8. http://www.cs.toronto.edu/~fritz/absps/reluICML.pdf

9. https://arxiv.org/pdf/1703.07737.pdf

10. https://arxiv.org/pdf/1608.00859.pdf

11. https://arxiv.org/pdf/1703.09783.pdf

# Appendix

**Recommendations for Splitting Up a Training Budget:**
**Based on Number of Available Snapshot Cycles**

| **# of Snapshot Cycles in Training Budget** | **Suggested Division of Time** |
|---|---|
| 1 | One single Snapshot model |
| 2 | Ensemble with two Snapshot models (with weight reset between each) |
| 3 | Ensemble with three Snapshot models (with weight reset between each) |
| 4 | Ensemble with four Snapshot models (with weight reset only between 2nd and 3rd model) |
| 5 | Same as #4 (but with each Snapshot model taking longer to train) |
| 6 | Ensemble with six Snapshot models (with weight reset only between 3rd and 4th model) |
| 7 | Same as #6 (but with each Snapshot model taking longer to train) |
| 8 | Ensemble with six Snapshot models (with weight reset after both 2nd and 4th models) |
| 9 | Same as #8 (but with each Snapshot model taking longer to train) |
| 10 | Ensemble with four Snapshot models (train seven Snapshot models though with weight reset after both 2nd and 5th models and throw away 1st, 3rd, and 6th Snapshot models) |
| 11 | Ensemble with six Snapshot models (train nine Snapshot models though with weight reset after both 3rd and 6th models and throw away 1st, 4th, and 7th Snapshot models) |
| 12 | Same as #11 (but with each Snapshot model taking longer to train and throw away the 2nd, 5th, and 8th Snapshot models instead of the 1st, 4th, and 7th Snapshot models) |

*Figure A1. Table providing guidelines for splitting training time into appropriate number and sequence of Snapshot models. It is assumed that each Snapshot cycle takes about 1/6 of the time of a full training budget. It should also be pointed out that for the longer sequences especially, increasing the number of weight resets and underlying models no longer contribute to the ensemble accuracy. Instead, the length of training for each model is a better source of accuracy improvement. And finally, any training cycle can be replaced with one using an alternative loss function or input data if suitable for improved results.*

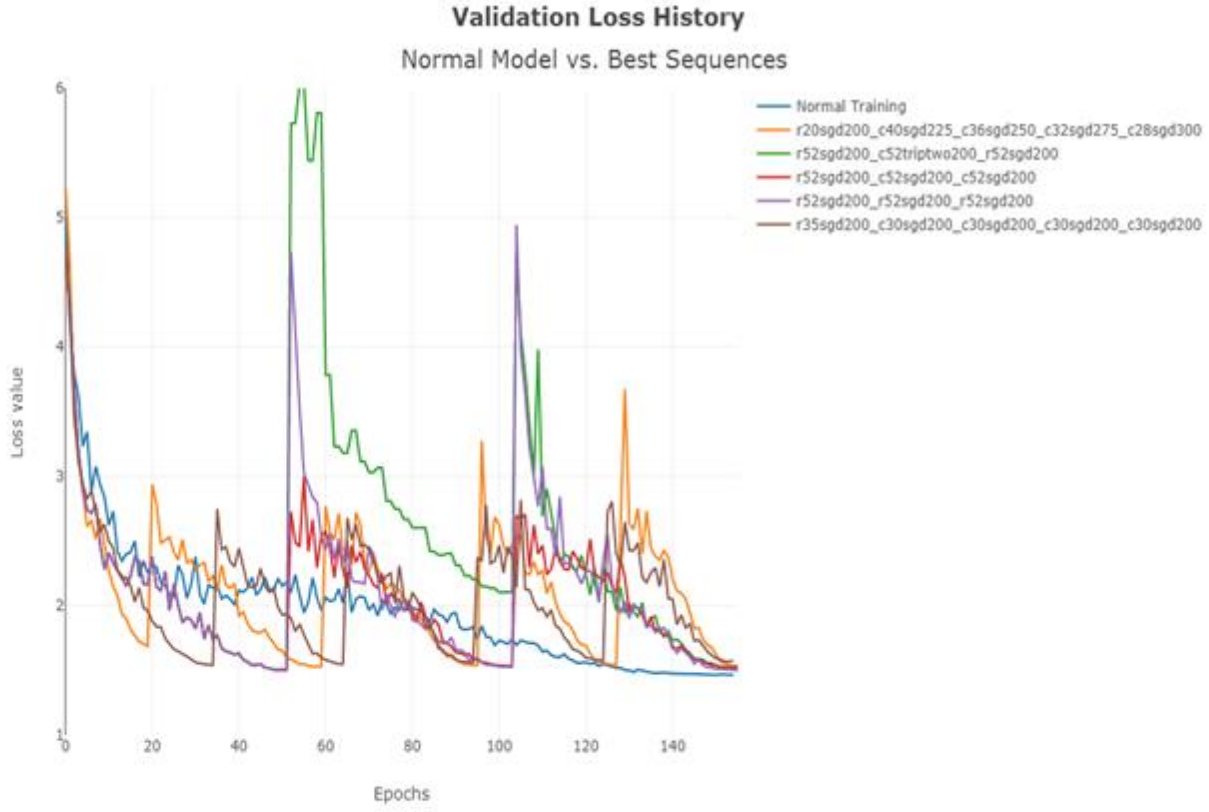


*Figure A2. Simple line plot comparing the validation loss during training of the various Snapshot ensemble solutions. There is not much analysis necessary here. Mostly, this graph is for edification purposes to demonstrate training differences between the proposed Snapshot ensembles and a normal training regimen.*

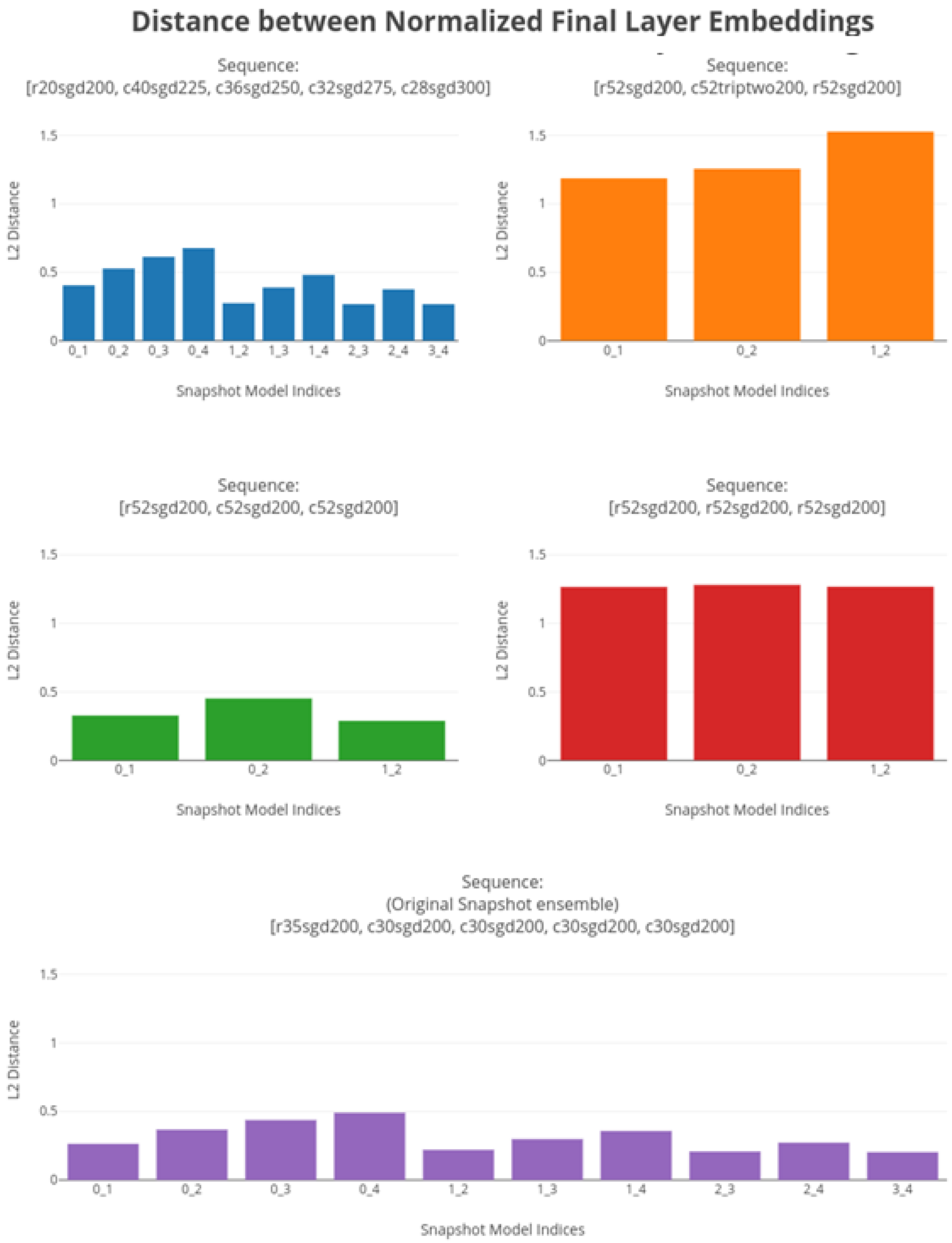


*Figure A3. Comparison of the final layer embedding distances between Snapshot models for an ensemble. Typically, the ensembles with more weight resets (denoted by the letter "r" at the beginning of a sequence token) will have larger distances between their embeddings and more variety in their local minima.*

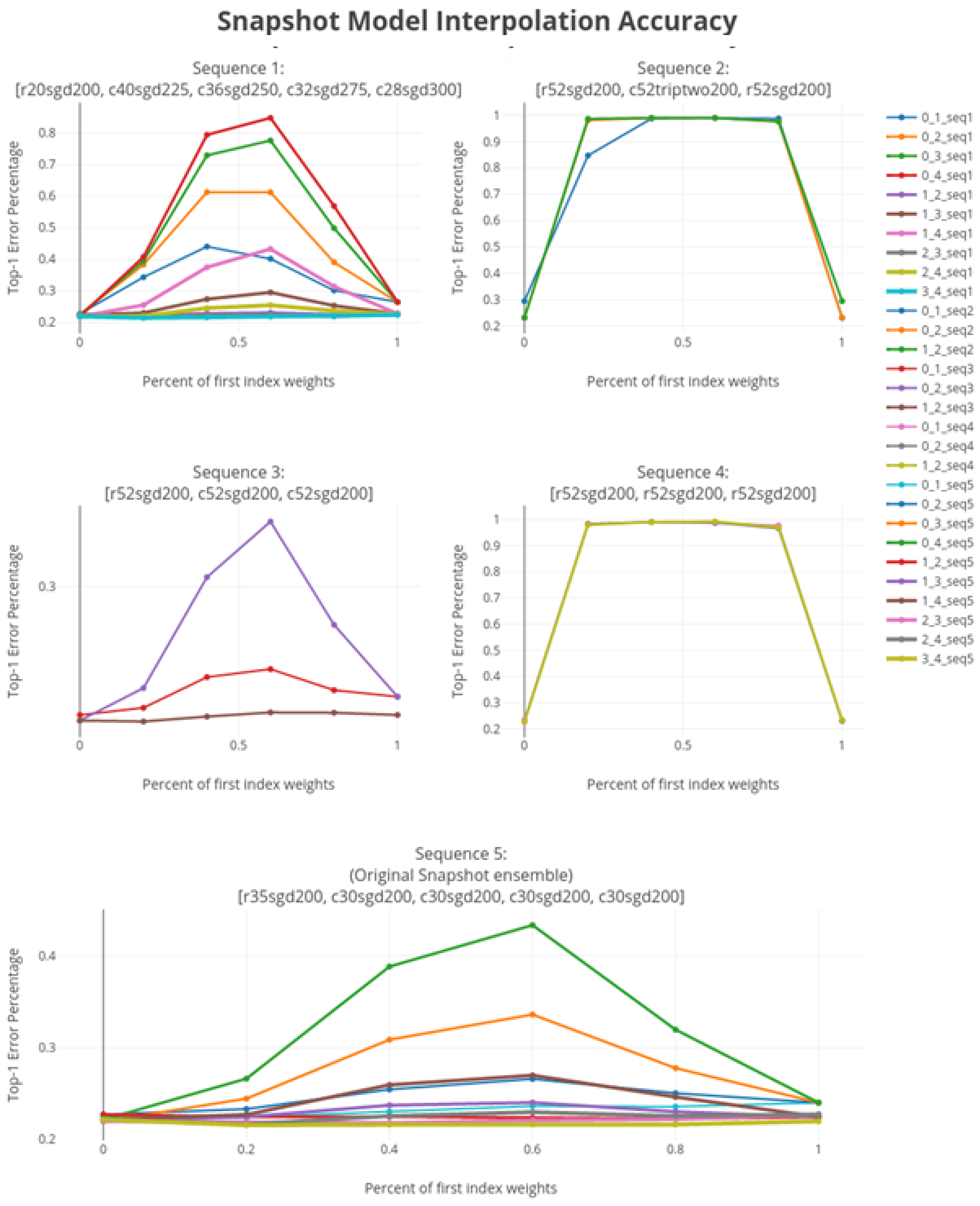


*Figure A4. Demonstration of a Snapshot model's ability to linearly mix its weights with another model within the ensemble. Generally, two models capable of their mixing weights without experiencing a large jump in error rate have settled at similar local minima.*

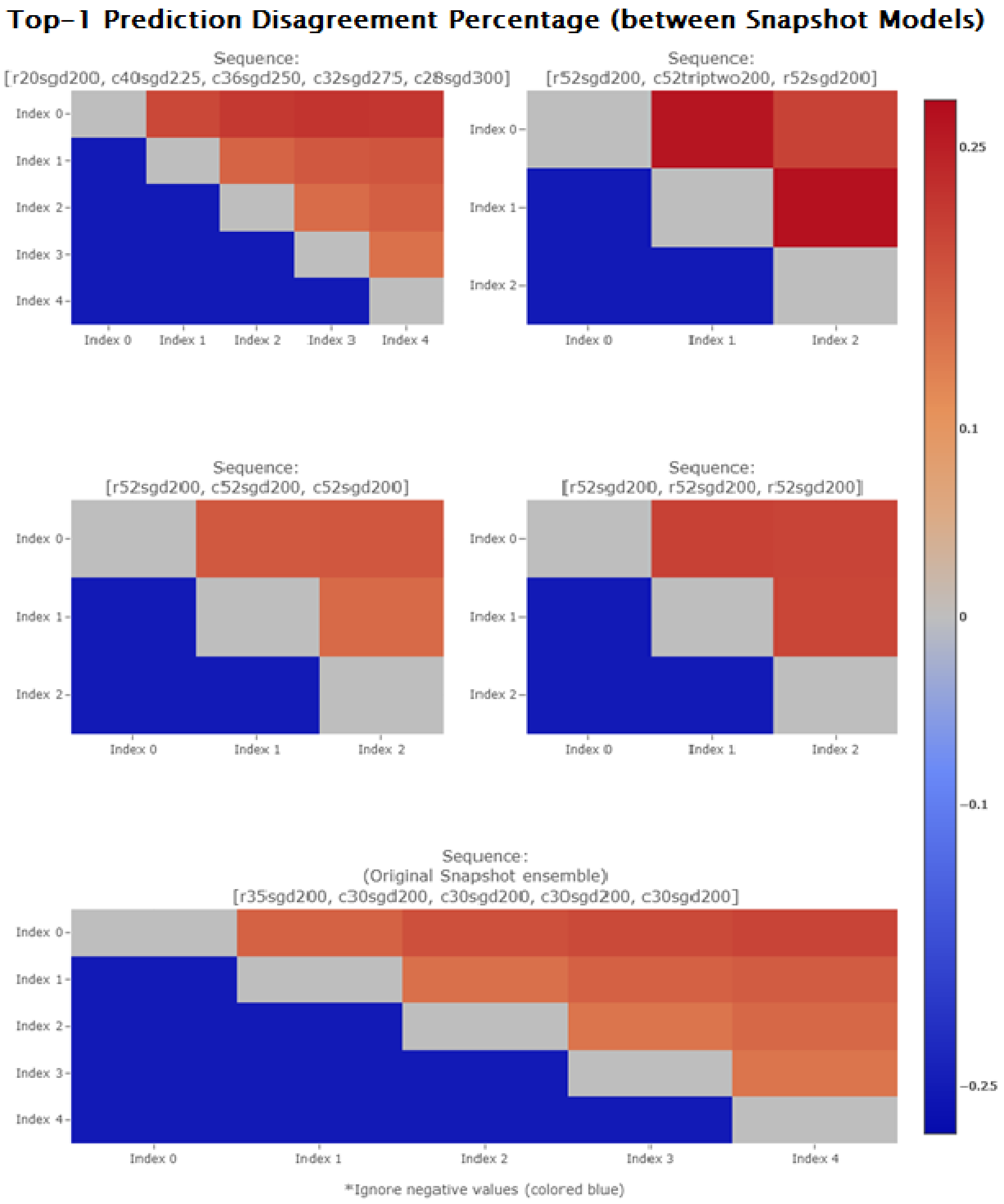


*Figure A5. Heat maps showing the prediction disagreement percentage between underlying models of an ensemble. Assuming its underlying model accuracies remain constant, an ensemble almost always benefits from more disagreement between its models (in terms of a higher overall accuracy for the ensemble).*

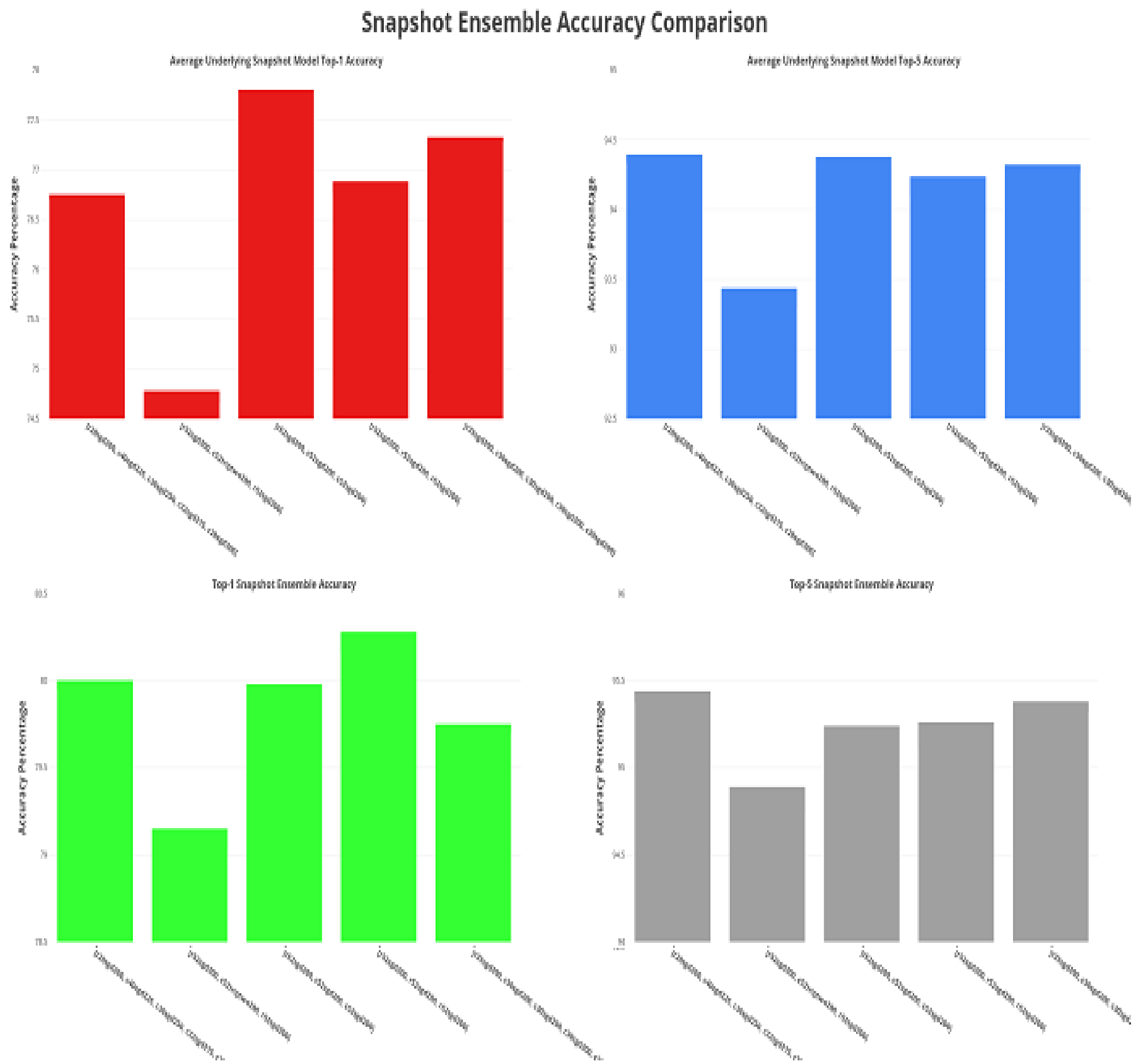


*Figure A6. Bar charts to highlight differences in the final evaluation metrics of the underlying models and overall ensembles of the proposed alternatives to the original Snapshot ensemble.*

.